CAIRO: A Compiler-Assisted Technique for Enabling Instruction-Level Offloading of Processing-In-Memory

RAMYAD HADIDI, LIFENG NAI, HYOJONG KIM, and HYESOON KIM,
Georgia Institute of Technology

Three-dimensional (3D) stacking technology and the memory-wall problem have popularized processing-in-memory (PIM), which offers the benefits of bandwidth and energy savings by offloading computations to functional units inside the memory. Several memory vendors have also started to integrate computation logics into the memory, such as Hybrid Memory Cube (HMC), the latest version of which supports up to 18 in-memory atomic instructions. Although industry prototypes have motivated studies for investigating efficient methods and architectures for PIM, researchers have not proposed a systematic way for identifying the benefits of *instruction-level PIM offloading*. As a result, compiler support for recognizing offloading candidates and utilizing instruction-level PIM offloading is unavailable. In this paper, we analyze the advantages of instruction-level PIM offloading in the context of HMC-atomic instructions for graph-computing applications and propose *CAIRO*, a compiler-assisted technique and decision model for enabling instruction-level offloading of PIM without any burden on programmers. To develop CAIRO, we analyzed how instruction offloading enables performance gain in both CPU and GPU workloads. Our studies show that performance gain from bandwidth savings, the ratio of number of cache misses to total cache accesses, and the overhead of host atomic instructions are the key factors in selecting an offloading candidate. Based on our analytical models, we characterize the properties of beneficial and non-beneficial candidates for offloading. We evaluate CAIRO with 27 multi-threaded CPU and 36 GPU benchmarks. In our evaluation, CAIRO not only doubles the speedup for a set of PIM-beneficial workloads by exploiting HMC-atomic instructions but also prevents slowdown caused by incorrect offloading decisions for other workloads.

CCS Concepts: •**Hardware** → *3D integrated circuits; Emerging architectures; Memory and dense storage;* •**Software and its engineering** → *Compilers;*

Additional Key Words and Phrases: Processing In Memory, Instruction-Level Offloading, Compiler Technique, Profiling, Hybrid Memory Cube (HMC), Emerging Technologies

ACM Reference format:
Ramyad Hadidi, Lifeng Nai, Hyojong Kim, and Hyesoon Kim. 2017. CAIRO: A Compiler-Assisted Technique for Enabling Instruction-Level Offloading of Processing-In-Memory. *ACM Transactions on Architecture and Code Optimization* 1, 1, Article 1 (January 2017), 24 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In modern computing systems, processors have steadily become more computationally powerful and energy efficient. However, the bandwidth, latency, and energy consumption of off-chip memories have not improved at the same rate as processors [32, 39], leading to the *memory-wall* [43] issue. To address this issue, a few decades ago, researchers proposed the *processing-in-memory (PIM)* technique [13, 15, 20, 31, 34], which reduces the overhead of data movement between off-chip

DOI: 10.1145/nnnnnnn.nnnnnnn

memories and processors by placing computations near data and utilizing the abundant in-memory bandwidth. Recently, the concept of PIM has been revisited because of the advances in 3D-stacking technology [9, 19, 36] and the prevalence of data-intensive applications, such as graph-computing applications [28]. Recent academic studies have proposed several PIM architectures and programming models [2, 3, 12, 17, 21, 22, 26, 44]. Memory vendors have also begun to incorporate PIM techniques such as *Hybrid Memory Cube (HMC)* [19, 35], *High Bandwidth Memory (HBM)* [23, 25], and *Active Memory Cube (AMC)* [29]. Among the existing academic and industrial proposals, HMC is a real-world PIM design that enables instruction-level offloading to the memory with 18 atomic computation instructions (noted as *HMC-atomic instructions*) starting with HMC 2.0[1] [8]. In other words, HMC-atomic instructions enable host to offload simple computations to HMC with an instruction-level granularity.

Instruction-level offloading to the memory eliminates unnecessary data movement between host processors and memories, and potentially offers both performance and energy benefits. However, blindly offloading any candidate instructions is not always beneficial. For example, offloading instructions with high data locality usually degrades performance. For instance, in a study by Ahn et al. (*PEI*) [3], a hardware-based locality monitor chooses a processing unit, either in the host processor or memory hierarchy, for the execution of each custom PIM instruction written by a programmer. In another study, Nai et al. (*GraphPIM*) [26] showed offloading with HMC-atomic instructions on CPU is beneficial for graph-computing applications. Although, comparing to PEI, GraphPIM suggests a design based on a real-world hardware specification and less complexity, similar to PEI, GraphPIM *does not explore candidate properties and selection process*. In other words, to achieve the full potential of instruction-level offloading, identifying proper offloading candidates without a burden on programmers is crucial, a task which in both PEI and GraphPIM is performed by a programmer. However, because of the complexity of modern applications, recognizing offloading candidates is not a trivial task. Therefore, providing compiler support that automates and facilitates the selection of offloading candidates is necessary.

In this paper, a follow up to GraphPIM, *we extend GraphPIM to GPU workloads* and introduce a compiler-assisted technique that facilitates instruction-level offloading on both CPU and GPU platforms in the context of HMC-atomic instructions for graph-computing applications. In particular, the challenges of enabling offloading for HMC-atomic instructions include (i) recognizing all applicable candidates whether beneficial or not when offloaded to the memory, (ii) identifying sources of instruction-level offloading benefits and their impact on performance and bandwidth, and (iii) selecting beneficial offloading candidates that is compatible with of HMC-atomic instructions. In the context of the HMC 2.0 design, we first analyze and model sources of instruction-level PIM offloading benefits such as cache bypassing for low-locality data, enabling memory bandwidth saving, and avoiding the overhead of host atomic instructions. Then, we propose *CAIRO*, a **C**ompiler-**A**ssisted technique and decision model for enabling **I**nst**R**uction-level **O**ffloading for PIM for both CPUs and GPUs during compilation time without any burden on programmers. While we explore instruction-level PIM offloading in the context of HMC-atomic instructions, we can apply the proposed technique to other instruction-level PIM offloading methods. Furthermore, although we investigate graph-computing applications because of their inefficient execution on modern systems, our analysis is also applicable to various applications. The key contributions of our paper are as follows:

- We propose the first compile-time technique that selects PIM candidates for instruction-level offloading in both CPU and GPU platforms.

[1] HMC 2.0 [8] differs from HMC Gen2 that follows the HMC 1.1 specification [16]. HMC 2.0 follows a newer specification [8]. HMC 2.0 hardware is not publicly available.

- We conduct a unified study that ascertains benefits of instruction-level offloading for PIM on both host platforms of CPU and GPU.
- We propose analytical models for estimating the benefits of instruction-level PIM offloading.
- We evaluate CAIRO by simulating 27 multi-threaded CPU and 36 GPU workloads. CAIRO shows up to 2x speedup for beneficial workloads and no slowdown for non-beneficial workloads.

The rest of this paper is organized as follows. Section 2 provides the background on PIM approaches, HMC, and HMC-atomic instructions. Section 3 describes instruction-level PIM offloading benefits. Section 4 presents our compiler technique, and Section 5 summarizes our evaluation methodology and results. Section 6 illustrates a case study, and Section 7 summarizes previous PIM research. Section 8 concludes the paper.

2 BACKGROUND

2.1 Instruction-Level PIM Offloading

Offloading computations to PIM units covers a broad range, of which, in this paper, we examine instruction-level offloading to fixed functions. In instruction-level offloading, the granularity of offloading is a instruction, and PIM units perform fixed-function operations; whereas, in kernel- or function-level offloading, the granularity of offloading is a whole kernel or a custom function such as scatter/gather, and PIM units perform more complex operations. One of the main differences between the instruction- and kernel-level offloading is whether the memory controller maintains a program counter. In the instruction-level offloading, all the offloaded computation is sent by extending memory commands. Therefore, the memory controller does not require any program counter or capability to access to the instructions of the program. On the other hand, for the kernel-level offloading, the memory controller maintains a program counter and must be able to fetch instructions.

Although each approach has its own restrictions, advantages, and benefits, we focus on instruction-level offloading, the benefits of which are saving the bandwidth [27], reducing the latency [27], increasing the effective size of system caches, avoiding the overhead of host atomic instructions [26], exploiting memory-level parallelism, designing an efficient and practical hardware [3, 14, 26], and increasing programmability [14, 26]. For instance, when we offload an host atomic instruction that its parameters does not have locality to PIM; first, we save bandwidth because we do not fetch the parameters of the instruction from the memory subsystem, instead we transmit the instruction, the size of which is smaller than that of the parameters. Second, the latency decreases because instead of brining the data from the memory, performing the operation in the processor, and writing it back again to the memory; we only transmit the instruction (in some cases we might also need an acknowledgment). Third, if the parameters are cache unfriendly, by offloading, we have increased the effective size of the cache. In this paper, we examine these benefits and identify how they correlate with the characteristics of instructions, and then we propose a compiler-assisted technique for identifying instruction-level offloading candidates.

2.2 Hybrid Memory Cube (HMC)

HMC is a collection of DRAM dies with a CMOS *logic die* fabricated with 3D-stacking technology. The dies are vertically connected by several *through-silicon vias (TSVs)* that provide higher internal bandwidth, lower latency, and lower communication energy consumption within a cube than two-dimensional (2D) organizations [19, 35, 44]. Each die is divided into equal *partitions*, and a group of vertically connected partitions with a memory controller in logic dies is a *vault*. To interface with the host processor, conventional DDRx uses the bus-based JEDEC protocol, whereas HMC uses its own communication packet, *Flit*. Flit-based communication enables HMC to use serialization

and deserialization (*SerDes*) circuits and reach higher external bandwidth of 480 GB/s according to the HMC 2.0 specification [8]. Starting with HMC 2.0 (with 32 vaults), HMC supports 18 atomic instructions in addition to conventional main memory tasks. For an HMC-atomic instruction, the memory controller of the corresponding vault basically performs three steps of operation: reads a data from a DRAM location, performs a computation on the data, and then writes back the result to the same DRAM location. According to the specification, memory controllers (i.e., PIM units) perform these steps, or read-modify-write (RMW) operation *atomically* within an HMC package. That is, the corresponding DRAM bank is locked during the RMW operation, so requests to the same bank must wait and cannot be serviced. Moreover, all PIM operations must include only one memory operand, or RMW operations are performed on an immediate value and a memory operand. Table 1 shows HMC-atomic instructions supported by HMC 2.0: Arithmetic, bitwise, Boolean, and comparison instructions. While the data size of these instructions are 16 bytes, some operations also support eight bytes. Depending on the definition of a specific instruction, a response may or may not be returned. If the response is returned, it includes an atomic flag that indicates whether the atomic operation was successful. Also, depending on the instruction, the original data (i.e., data before modification) may also be returned along with the response.

Table 1. HMC-atomic instructions in HMC 2.0 [8].

Type	Data Size	Operation	Return
Arithmetic	8/16-byte	single/dual signed add	w/ or w/o
Bitwise	8/16-byte	swap, bit write	w/ or w/o
Boolean	16-byte	AND/NAND	w/o
		OR/NOR/XOR	w/o
Comparison	8/16-byte	CAS-if equal/zero	w/
		greater/less	w/
		compare if equal w/	

2.3 Supporting HMC-Atomic Instructions

2.3.1 Candidates for Offloading. As described in Section 2.2, HMC-atomic instructions perform RMW atomic operations on a single address. An offloading candidate is one instruction or a group of instructions that can be converted to HMC-atomic instructions. In general, HMC-atomic instructions support two types of offloading candidates: (A) a group of instructions in a single thread that perform RMW from/to a single location and (B) generic host atomic instructions. For the first type, a group of instructions must contain the following instructions: (i) a single load instruction reading data λ from a memory address, (ii) a simple compute operation on λ that can be mapped to an HMC-atomic instruction, and (iii) a single store instruction writing back λ to the memory address. The second type of offloading candidates are generic host atomic instructions that inherently perform the aforementioned operations such as *CMPXCHG* in *x86*, *SWP* in *ARM*, and *atomicCAS* in *PTX*.[2]

2.3.2 Burden on Programmers. To identify and offload HMC-atomic instructions, the underlying architecture must identify offloading candidates that CAIRO detected during compilation. To carry out this task, we introduce two methods. The first method, similar to GraphPIM [26], is to mark a memory region as the *HMC offloading region* and map the host atomic instructions to the corresponding HMC-atomic instructions in that region during execution. Thus, any host atomic instruction that accesses the region will be offloaded to the PIM. The second method, similar to PEI [3], is to employ an ISA extension in the host processor for each HMC-atomic instruction.

[2]Some host atomic instructions enforce fence semantics; therefore, we add dependency to the return value of the converted HMC-atomic instruction.

Without a compiler support, both methods place the burden of selecting offloading candidates or regions on the programmer. In fact, because identification of proper offloading candidates requires careful consideration of multiple factors such as cache performance, bandwidth consumption, and application behavior, this burden limits PIM applicability. In some cases, HMC-atomic instructions can violate memory consistency, but as pointed out in [24, 26], graph-computing applications can safely use HMC-atomic instructions. This is because these applications perform read and atomic instructions at separate execution phases, which naturally prevents consistency violation. Section 6 provides an example of such applications. Therefore, to extend HMC-atomic instructions applicability to other applications, the only burden on a programmer is ensuring memory consistency when instructions are offloaded to HMC.

2.3.3 Cache Policy. Because HMC-atomic instructions directly modify data within the HMC, we maintain a *cache-bypassing* policy that ensures a coherent view of offloading targets. In other words, marking the memory accesses of HMC-atomic instructions as uncacheable causes them to bypass the cache hierarchy and ensures that a single copy of offloading targets exists. Maintaining a cache-bypassing policy, rather than full coherence caches is also more affordable in terms of design complexity. (Note that only HMC-atomic instructions bypass caches, the rest of instructions use the unmodified cache policy of the host.) In fact, following this policy, a programmer should allocate irregular data all together to minimize the negative effect of uncaching cache-friendly data. In this paper, irregular data are allocated altogether, similar to GraphPIM [27]. To support the cache-bypassing policy, we can utilize existing uncacheable (UC) memory support in x86 architectures [18]. This feature marks corresponding physical pages as uncacheable by setting system registers (such as MTRRs [18]) through the operating systems [26]. In addition to x86, to support the cache-bypassing policy, other architectures should support a similar feature. The only difference between sparse and allocated data in CAIRO would be maintaining a fine-grained coherence cache policy that increases the complexity. The performance overhead as a result of such complexity depends on the exact coherency mechanism of the underlying architecture, which is an orthogonal problem. In fact, allocating irregular data together simplifies big-data programs, and it is a common practice [26].

2.3.4 HMC-Atomic Instruction Extensions. The current HMC specification supports up to 18 simple HMC-atomic instructions with the signed addition operation as the most complex. In our simulation, each operation in the ALU of a vault takes three cycles. During the operation (i.e., read, modify, and write), the corresponding DRAM bank is locked to ensure consistency. Also, additional RMW requests to a vault will be queued in a FIFO buffer within the vault to be serviced when the ALU is free and the corresponding bank has no outstanding requests. (In our studies, the required buffer size did not exceed ten entry per vault.) To determine possible extensions to current HMC-atomic instructions, we have also added a low-cost, high-latency, single-precision (32-bit) floating-point-addition instruction to the HMC-atomic instructions. We integrate a floating-point addition to the ALU within each vault with 40-cycle latency. Such integration has been done before [3, 26], and we verified that its implementation does not violate any power or area constraints.

2.4 Graph-Computing Applications

Several real-world computing problems employ graphs (e.g., social networks and web graphs) for processing the large-scale of network data. The diversity and unique characteristics of graph-computing applications causes different computation behaviors such as: Graph traversals with a large number of irregular memory accesses, rich property graphs with heavy computations similar to regular applications, and dynamic graphs with irregular memory accesses and dynamic memory footprint [28].

In this paper, we examine graph-computing applications because of their inefficient execution on modern systems and their potential for PIM offloading [2, 26]. We extend GraphPIM [26] to GPU workloads and propose CAIRO for identifying offloading candidates for both CPU and GPU workloads without any burden on programmers. Although we investigate graph-computing applications, the proposed technique and methodology is applicable to other applications. We leave such thorough analysis to future work, and focus on graph-computing applications in this paper.

3 BENEFITS OF OFFLOADING

3.1 Bandwidth Savings

Instruction-level offloading to the HMC saves bandwidth for three reasons: (i) we do not fetch unnecessary cache-line data from memory, (ii) we issue one HMC-atomic instruction instead of issuing one load and one store instruction to the memory, and (iii) we might save more bandwidth because we would prevent future coherence messages regarding the data of the cache line. We ensure offloading targets are cache-unfriendly; therefore, bringing their data into caches is redundant and wastes system resources. In our technique, we ensure that a candidate is cache unfriendly by profiling its LLC (last level cache) miss ratio that is defined as $MR_{LLC} = {}^{\text{Number of LLC misses}}/_{\text{Number of LLC accesses}}$.

Table 2. Bandwidth requirements of HMC transactions in flits (Flit size: 16 bytes, or 128 bits) [8].

	Type	Request	Response	Total
Reg.	64-byte READ	1 flit	5 flits	6 flits
	64-byte WRITE	5 flits	1 flit	6 flits
Atomic	add without return	2 flits	1 flit	3 flits
	add with return	2 flits	2 flits	4 flits
	Boolean/bitwise/CAS	2 flits	2 flits	4 flits
	compare if equal	2 flits	1 flit	3 flits

To estimate the bandwidth savings from the offloading of HMC-atomic instruction, first, we describe the communication protocol of HMC, and then we provide an analytical model for off-chip bandwidth. HMC follows a packet-based communication protocol; a packet consists of 16-byte flow units called Flit [8]. Table 2 summarizes the packet sizes for regular memory requests and HMC-atomic instructions. Now, assume an application with a single offloading candidate. Let N_c be the number of times that the application executes the candidate. Also, let N_{reg} be the number of times that the application executes regular memory requests. Assume α and β are the average LLC hit ratio (i.e., $1 - MR_{LLC}$) of the offloading candidate and regular memory requests, respectively. In a case without instruction-level offloading support, both the candidate and regular memory requests and responses (read, or write) will consume 6 flits (See Table 2). Therefore, the regular bandwidth usage (BW_{reg}) for the application is

$$BW_{\text{reg}} = \big(2N_c(1-\alpha)\times 6\big) + \big(N_{\text{reg}}(1-\beta)\times 6\big) \text{ flits.} \quad (1)$$

BW_{reg}: Regular bandwidth usage (without offloading)
N_c: Number of the candidate execution
N_{reg}: Number of the regular memory requests execution
α: Average LLC hit ratio of the candidate
β: Average LLC hit ratio of regular memory requests

In this equation, each execution of the offloading candidate contains one read and write instructions, so we multiply N_c by two. When we perform a single HMC-atomic instruction, we typically require

three or four flits. By assuming a total packet size of δ flits for a single execution of the offloading candidate (request and response), the bandwidth usage with offloading ($BW_{offload}$) is

$$BW_{\text{offload}} = \big(N_c \times \delta\big) + \big(N_{\text{reg}}(1-\beta) \times 6\big) \text{ flits.} \quad (2)$$

BW_{offload}: Bandwidth usage with offloading
δ: Total packet size of request and response for offloading a candidate in flits

Therefore, bandwidth savings with offloading is

$$\begin{aligned} BW_{\text{saving}} &= BW_{\text{reg}} - BW_{\text{offload}} \\ &= 12N_c(1-\alpha) - \delta N_c = N_c\big(12(1-\alpha) - \delta\big) \text{ flits} \\ &= \begin{cases} 3N_c(3-4\alpha) \text{ flits}, & \text{if } \delta = 3 \\ 4N_c(2-3\alpha) \text{ flits}, & \text{if } \delta = 4. \end{cases} \end{aligned} \quad (3)$$

Figure 1 illustrates situations in which offloading saves bandwidth for two different δ values. As shown, the cache hit ratio and offloading type of a candidate determine the amount of bandwidth savings. In fact, the cache hit ratio for various candidates depends on the bandwidth requirements, the system cache hierarchy, and the offloading target localities of each. Our simple analytical model shows that the candidate locality and its type are key factors in the amount of bandwidth savings.

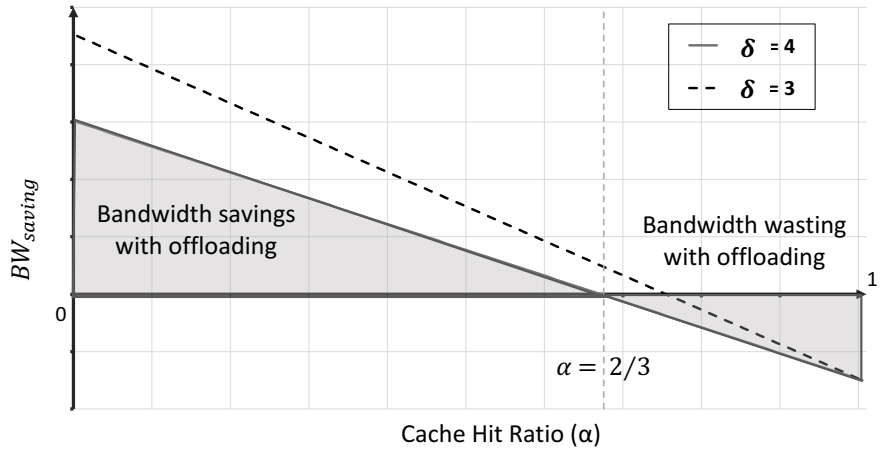

Fig. 1. Difference between bandwidth use of no offloading and offloading cases with various candidate cache hit ratios and required packets for the target HMC-atomic instruction (δ).

3.2 Bandwidth-Savings Performance Benefits

Increased bandwidth available to the memory has a positive impact on the performance of an application for two reasons: (i) each memory request has shorter latency because of the reduced queuing delay, and (ii) the application can issue more memory requests in a cycle. From a system perspective, both reasons are translated into shorter latency of memory accesses. However, from an application perspective, if issuing more memory requests is currently limited by the low bandwidth of memory, providing more memory bandwidth leads to improvement in performance (*bandwidth-sensitive* applications) because these types of applications exploit *memory-level parallelism (MLP)*. For applications such as compute-intensive applications, however, enabling more bandwidth has a small impact on performance (*bandwidth-insensitive* applications).

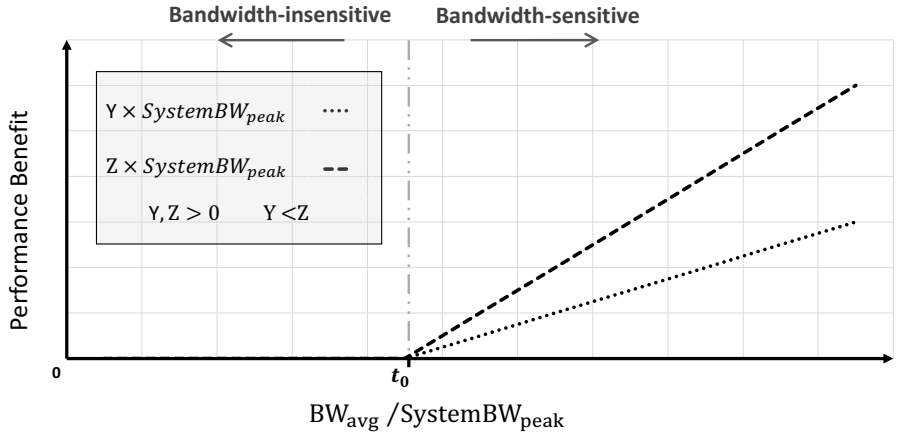

Fig. 2. Analysis of performance and bandwidth-savings: Bandwidth-sensitive applications (right side of the graph) gain performance with increased bandwidth.

Offloading using HMC-atomic instructions provides more available bandwidth to applications. Bandwidth-sensitive applications gain a major performance benefit by exploiting free bandwidth enabled by bandwidth savings, but this is not true for bandwidth-insensitive applications, as Figure 2 depicts. This conceptual graph shows the performance benefit when the bandwidth of a system increases by a factor of Y and Z. The lines represent a collection of applications with various average bandwidth utilizations that spans in the range of zero to the peak bandwidth of the system ($SystemBW_{\text{peak}}$). We define average bandwidth of an application with $BW_{\text{avg}} = (n_{\text{access}} \times \text{size}_{\text{access}})/t_{\text{exe}}$, in which n_{access} is the number of accesses, $\text{size}_{\text{access}}$ is the average size of accesses, and t_{exe} is the application execution time. Bandwidth-sensitive applications naturally have higher average bandwidth utilization than bandwidth-insensitive applications, so they reside on the right side of the graph, where $BW_{\text{avg}}/SystemBW_{\text{peak}}$ is high. If we increase $\text{SystemBW}_{\text{peak}}$, bandwidth-sensitive applications experience a performance gain. While, bandwidth-insensitive applications are not susceptible to the bandwidth increment. In other words, offloading provides bandwidth savings, which has the same effect as increasing $SystemBW_{\text{peak}}$. If the benchmark is on the left side of the graph (i.e., the bandwidth of the application is less than a threshold), offloading will not result in any performance benefit; thus, the offloading decision must ensure the required bandwidth for the application exceeds a certain threshold. In the figure, this threshold value is t_0. The exact value of this threshold depends on the application, architecture, and programming model. In Section 5.2, we will see that CPU workloads tend to be on the left side of the graph whereas GPU workloads tend to be on the right side.

3.3 Cache-Related Benefits

Executing HMC-atomic instructions increases the effective cache size by offloading cache-unfriendly data. We ensure that the offloading candidate is cache-unfriendly by using a cache-profiling tool. As pointed out in Section 2.3.3, the data of offloaded HMC-atomic instructions are identified as uncacheable, so they bypass the cache hierarchy and allow more effective use of cache space. Bypassing the cache also prevents the overhead of unnecessary cache-checking time for the data of offloading candidates, as GraphPIM [26] showed with a detailed breakdown of applications execution. We will discuss these points in more detail in the evaluation section.

3.4 Preventing The Overhead of Host Atomic Instructions

In ARM and x86 architectures, generic atomic instructions incur substantial overhead because of their consistency and ILP restrictions [26, 42]. Moreover, AMD and NVIDIA GPU architectures contain this overhead [10, 33]. To ascertain the extent of the overhead in atomic instructions, similar to Nai's evaluation [26], we conducted a real machine experiment on an Intel Xeon E5-2620 using graph-processing kernels. We create micro-benchmarks that perform one iteration of each kernel using either generic atomic instructions (atomic) or regular instructions (non-atomic) and then execute both versions.[3] Figure 3 shows kernels that use host atomic instructions suffer 23% more on average in speedup than kernels that use regular reads and writes. This overhead is mainly because atomic instructions stalls the pipeline and drains write buffer for ensuring consistency. Furthermore, the accessed data could be in the shared state in other processors, so an another source of the overhead is due to cache invalidation and coherence traffic. Because HMC-atomic instructions exploit MLP, and the programming model of target applications does not require strict sequential consistency (*SC*)[4], the execution of atomic operation will have less overhead than their execution on a host [24, 26, 27].

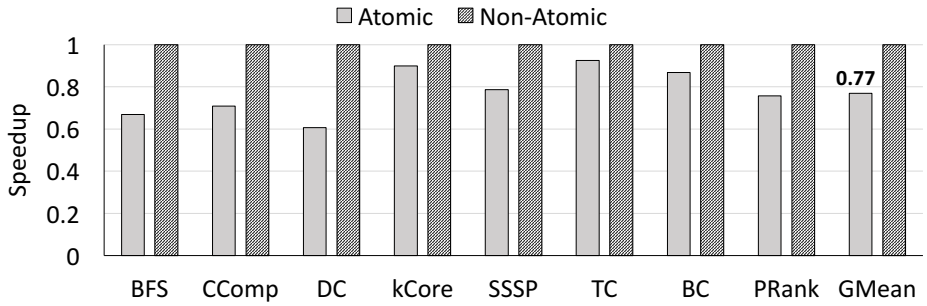

Fig. 3. The degradation of speedup for host atomic instructions in graph-processing kernels on an Intel Xeon E5-2620 machine.

4 COMPILER HEURISTICS

In this section, we develop heuristics and propose CAIRO, a profiling-based compile-time technique for instruction-level PIM offloading. To select proper offloading candidates, CAIRO examines the application for eligible candidates, estimates the cache miss ratio of each candidate with a cache profiler, and preforms analyses to make offloading decisions.

4.1 Bandwidth-Insensitive Applications Analysis

HMC-atomic instructions enable the benefits of cache bypassing, and remove the overhead of host atomic instructions for bandwidth-insensitive applications, as Section 3 discussed. To determine the impact of cache bypassing on performance, a key factor is the LLC miss ratio of offloading candidates. Bypassing the cache for a candidate with a low cache miss ratio causes performance degradation whereas offloading for a candidate with a high cache miss ratio provides these benefits without the negative effect of cache bypassing. In fact, because of the low-locality of offloaded data, each execution of a candidate in the non-offloading case incurs two high-latency accesses to the

[3]Note that read and atomic operations phases in micro-benchmarks occur at two different program phases, so naturally the consistency and correctness issue is avoided in one iteration.

[4]To support applications that demand SC (not covered in this paper), HMC-atomic instructions require the insertion of memory barriers. Such insertions requires a support in either of the processor or HMC architecture, both of which demand a new hardware design.

memory (one read and one write-back). In other words, offloading hides high-latency accesses by exploiting MLP and provides more bandwidth to the application. Let us assume an application has one eligible candidate for offloading. If the candidate has a higher miss ratio, the application speedup gain by offloading it would be higher than the same candidate with a lower miss ratio. In Figure 4, we show a first-order approximation of the relationship between the LLC miss ratio of an offloading candidate and its performance speedup (the offloading over non-offloading case). In this relationship, MR_{th} is the cutoff miss ratio, which is lower than some miss ratios, with which the offloading of the candidate improves speedup. For bandwidth-insensitive applications, we can approximately represent the speedup (SU_{MR}) and the miss ratio of a candidate (MR) as $SU_{MR} = f(\mathrm{MR})$, where $f(x)$ defines a linear relationship. Furthermore, MR_{th} is determined when SU_{MR} equals to zero.

Another benefit of offloading for bandwidth-insensitive applications is preventing the overhead of host atomic instructions. The amount of speedup gain from removing this overhead depends on the density of host atomic instructions per memory region (ρ_{HA}). Therefore, a first-order approximation of speedup (SU_{HA}) from preventing this overhead is $SU_{HA} = g(\rho_{HA})$, where $g(x)$ defines a linear relationship. Now, since these two source of benefits, SU_{MR} and SU_{HA}, are independent, gained speedup (SU_{tot}) from offloading a candidate totals as

$$\begin{aligned} SU_{tot} &= \mathrm{SU}_{MR} + \mathrm{SU}_{HA} = f(\mathrm{MR}) + g(\rho_{HA}) \\ &= \big(C_1 \times MR\big) + \big(C_2 \times \rho_{HA}\big) + C_3, \end{aligned} \tag{4}$$

SU_{tot}: Total speedup from offloading (bandwidth-insensitive applications)
SU_{MR}: Speedup dependent to the miss ratio of a candidate
SU_{HA}: Speedup from avoiding the overhead of host atomic instructions
$f(x), g(x)$: Linear relationship functions
MR: LLC miss ratio of a candidate
ρ_{HA}: Density of host atomic instructions per memory region
C_1, C_2, C_3: Machine-dependent constants

where C_1, C_2, and C_3 are machine-dependent constants. In particular, C_2 depends on the architecture and its associated overhead for host atomic instructions. We measure machine-dependent constants with a single execution of a micro-benchmark on a machine. In Figure 4, we show the offloading speedup for the same application with a different density of host atomic instruction. Because

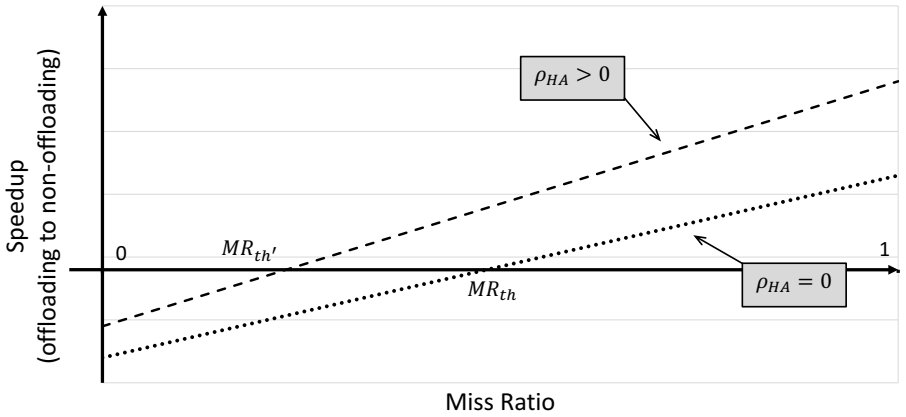

Fig. 4. Miss ratio of a candidate and the speedup of the offloading over non-offloading case for bandwidth-insensitive application variants with different generic host atomic instruction densities (ρ_{HA}). The variant with larger ρ_{HA} has smaller cutoff miss ratio ($MR_{th}^{'}$).

offloading also removes the overhead, the application variant with larger ρ_{HA} has a smaller cutoff miss ratio ($MR_{th}^{'}$) than the other variant. In CAIRO, for bandwidth-insensitive applications, we use Equation 4, or *miss-ratio analysis*, to check if offloading a candidate is beneficial.

4.2 Bandwidth-Sensitive Applications Analysis

In addition to the miss-ratio analysis, we should also determine the performance gain from bandwidth savings for bandwidth-sensitive applications in our offloading decision (see Section 3.2). Similar to bandwidth-insensitive applications, we employ the same approach based on the LLC miss ratio of the candidate to determine the speedup from offloading. Moreover, because bandwidth-sensitive applications also gain speedup from bandwidth savings, and the amount of bandwidth savings is directly dependent on the miss ratio of the candidate, such savings are important in the offloading decision. To analyze the gain, for a given cache miss ratio of a candidate, we have three decision regions, which are shown conceptually in Figure 5. If the miss ratio is lower than a certain threshold ($MissRatio_L$), because the candidate has a high cache locality, we do not offload the candidate. Also, if the miss ratio is higher than another threshold ($MissRatio_H$), since the performance benefits discussed in Section 4.1 is guaranteed, we offload the candidate. However, if the miss ratio is between the MissRatio$_L$ and the MissRatio$_H$, determining the offloading decision requires more analysis. In Figure 2, bandwidth-sensitive applications are on the right side of the diagram, so they experience more improvement in performance with more bandwidth savings. We model this behavior using normalized bandwidth savings ($BW_{\text{saving}}^{'}$) as

$$BW_{\text{saving}}^{'} = {}^{(BW_{\text{reg}} - BW_{\text{offload}})}/_{BW_{\text{reg}}} = {}^{BW_{\text{saving}}}/_{BW_{\text{reg}}}, \qquad (5)$$

$BW_{\text{saving}}^{'}$: Normalized bandwidth savings
BW_{reg}: Regular bandwidth usage (without offloading)
BW_{offload}: Bandwidth usage with offloading

where BW_{saving} and BW_{reg} is calculated as it was in Equation 3. Now, we represent speedup from the bandwidth savings with a first-order approximation as

$$SU_{BW} = \left(M_1 \times BW_{\text{saving}}^{'}\right) + M_2, \qquad (6)$$

SU_{BW}: Speedup from bandwidth savings (bandwidth-sensitive applications)
$BW_{\text{saving}}^{'}$: Normalized bandwidth savings

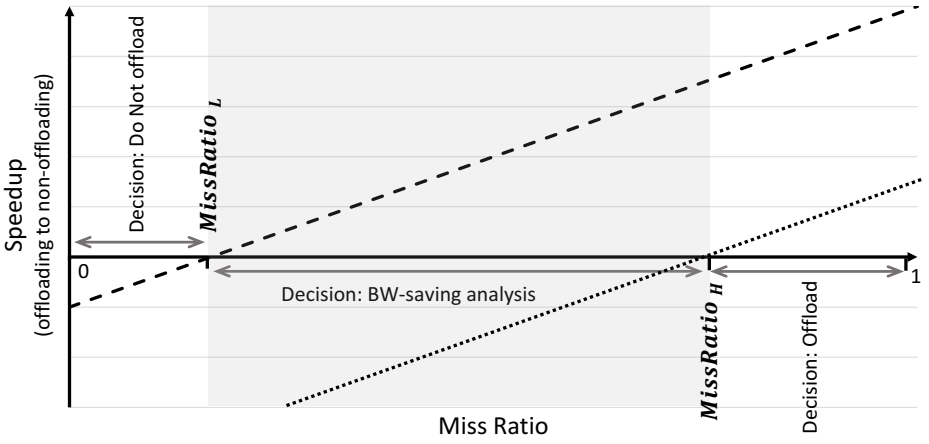

Fig. 5. Speedup of offloading versus non-offloading case and candidate miss ratio for bandwidth-sensitive applications.

where M_1 and M_2 are machine-dependent constants. We determine the performance gain from bandwidth savings by using Equation 6, or *BW-saving analysis*. In summary, for the bandwidth-sensitive applications if the miss ratio is between $MissRatio_L$ and $MissRatio_H$, we use Equation 6 for offloading decision. Otherwise, if the miss ratio is larger than $MissRatio_H$ we offload; and, if the miss ratio is smaller than $MissRatio_L$ we do not offload. In this paper, for $MissRatio_H$ and $MissRatio_L$ we use conservative values of 30% and 80%, respectively. A conservative estimation of these values increases false positives in performing BW-savings analysis, but it will avoid cases in which offloading the candidate hurts the performance. Note that described analysis is after analysis of Equation 4, and total speedup is calculated from the sum of SU_{BW} and SU_{tot}. However, for bandwidth-sensitive applications, SU_{BW} has the highest contribution in the total speedup.

4.3 Eligibility Test

HMC-atomic instructions support two types of offloading candidates: a group of instructions that perform RMW and host atomic instructions. Following the HMC specifications, we perform the *eligibility test*, which consists of a candidate density test that filters out low-density candidates, on the instructions of the application to select eligible offloading candidates. The eligibility test consists of the following:

a) Read-modify-store *(RMW)* test: Ensures that the outcome of converting an RMW operation on a single target is an HMC-atomic instruction, as pointed out in Section 2.3.1. Therefore, the existence of all of RMW operation components on the single target is essential. Moreover, this test also checks that offloading the RMW to PIM does not violate sequential consistency.
b) Simple-operation *(SO)* test: Ensures that HMC supports the operation performed on a candidate. Currently, HMC supports simple atomic operations (Table 1). As discussed in Section 2.3.4, we added the floating-point-addition instruction to the HMC-atomic instructions.
c) Direct-address limitation *(DAL)* test: Ensures that HMC-atomic instructions contain a direct memory address for read and write operations (i.e., the address of the operand is embedded in the instruction code or is accessible during execution); thus, load and store instructions must follow this limitation. For instance, `lw $1, ($10)` fails this test because the value of register `$1` is loaded from the memory location whose address is given by the contents of register `$10`.
d) Size-limitation *(SL)* test: Ensures that the size of the data on which an HMC-atomic instruction does computaiton is 16 , or eight bytes (see Section 2.2). Therefore, the data field cannot contain multiple variables or a larger variable than 16 bytes.
e) Candidate-density *(CD)* test: Ensures that the density of offloading candidates per memory region exceeds a certain amount so that the performance impact of their offloading will not be negligible. Note that this is the minimum requirement of the candidate density and we will still evaluate Equation 4 during miss-ratio analysis.

4.4 Summary of CAIRO

To select proper offloading candidates, CAIRO performs the steps shown in Figure 6. Before running CAIRO, we run a set of micro-benchmarks that ascertain machine-dependent constants for our analytical models and the host cache hierarchy. A micro-benchmark is a kernel that constitutes an offloading candidate with a configurable cache miss ratio. For each architecture, we run the micro-benchmark once to determine machine-dependent constants, which also can be provided by vendors. Therefore, the overhead of determining the machine-dependent constants is negligible. After that, we utilize a simple profiler that estimates the LLC miss ratio of memory accesses and

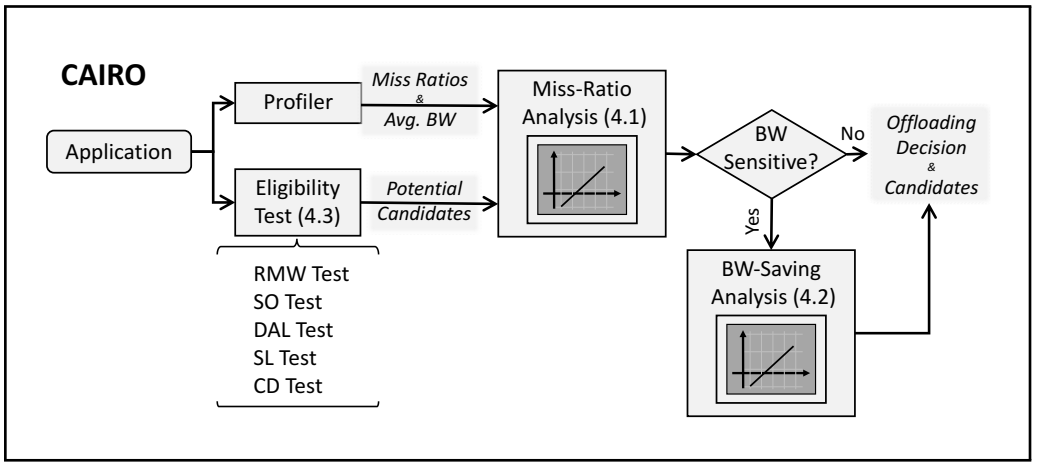

Fig. 6. Overview of CAIRO.

average bandwidth utilization of each candidate during the entire application runtime[5] (Average bandwidth is roughly calculated from the number of misses and the approximation of execution time.) Meanwhile, CAIRO also performs the eligibility test to identify eligible offloading candidates. Note that the eligibility test is the only step in CAIRO that depends on the details of the PIM hardware, and their implementations are possible with simple compiler passes. Then, CAIRO analyzes each candidate based on the analysis in Sections 4.1 and 4.2 (i.e., denoted "BW-savings analysis" and "miss-ratio analysis" in Figure 6). As discussed, candidates in bandwidth-sensitive applications might need an extra analysis for the selection. Finally, for each candidate and its associated memory region, CAIRO determines if offloading is beneficial and makes a decision.

5 EVALUATION

5.1 Evaluation Methodology

Because HMC 2.0 hardware is not publicly available, we evaluate CAIRO by performing detailed timing simulation. To simulate the host-side CPU/GPU architecture, we use MacSim [1], a cycle-level architecture simulator that supports both CPUs and GPUs. To simulate DRAM timings, network contentions, the HMC structure, and HMC-atomic instructions, we use an in-house 3D-stacked-memory simulator based on DRAMSim2 [41], VaultSim. We have validated the performance results of VaultSim using a real prototype of HMC 1.1. We use the Structural Simulation Toolkit (SST) [38] as the simulation framework for connecting MacSim and VaultSim. Table 3 shows the configuration of experiments. We model a processor with 16 out-of-order cores, a GPU with 15 SMs, and an 8 GB HMC that follows the specifications of HMC 2.0 [8, 19, 22, 35, 40].

In our evaluation, we use the benchmarks from GraphBIG [28], a comprehensive benchmark suite covering a wide scope of graph-computing applications for both CPU and GPU. Table 4 shows the selected benchmarks that passed the eligibility test.[6] To illustrate why CAIRO does not offload candidates for benchmarks with a low density of candidates (i.e., benchmarks that fail the CD test), we also include them in our evaluation. In brief, nine benchmarks are included with nine CPU workloads and 12 GPU workloads. Note that the same kernel in GPU(e.g., `BFS`) might have

[5]Similar to several studies that have utilized an integration of cache profiler with a compilation process, CAIRO can exploit tools such as Pin [37] or Perf to profile cache miss ratios. However, since applying offloading decisions to an application depends on an architecture and its characteristics, such integration, while an interesting compiler challenge, is beyond the scope of this paper.

Table 3. Simulation configurations for CPU+HMC and GPU+HMC systems.

Component	Configuration
CPU	16 x86 out-of-order cores, 2 GHz, 4-issue
Cache	32 KB private 4-way L1 data/instruction caches 256 KB private 8-way L2 inclusive cache 16 MB shared 16-way L3 inclusive cache 64-byte cache line - LRU - MESI
GPU	15 PTX SMs, 32 threads/warp, 1.4 GHz 16 KB private L1D, and 1 MB 16-way L2 cache
HMC	1 GHz, 8 GB cube, 32 vaults, 512 DRAM banks [8] $t_{CL} = t_{RCD} = t_{RP} = 13.75\ ns$, $t_{RAS} = 27.5\ ns$ [22] 4 links per package, 120GB/s per link [8]
ALUs	One ALU/vault with units described in Table 1, and one single-precision floating-point addition/vault
Computation Latency	HMC-atomic instructions: 3 cycles HMC Floating-point addition: 40 cycles

different algorithmic implementations. (e.g., `BFS-twc`, `BFS-dwc`, or `BFS-ttc` for GPU) [5, 28]. To better understand the impact of the cache miss ratio and bandwidth savings, we also generate new workloads with different cache miss ratios for their candidates by integrating a random variable in the simulator while maintaining the same program behavior. To distinguish these workloads, we annotated the end of the name of each workload with:"`-H`" for the original workload, "`-M`" for medium miss ratio setting, and "`-L`" for low miss ratio setting. We use the LDBC graph [11] (1M vertices, 900 MB memory footprint) as the input set of the benchmarks. We evaluate 27 and 36 workloads for CPUs and GPUs, respectively, shown in Table 4. In our evaluations, profiling is done once per workload. Similar to any profiling-based optimizations, selecting a representative input set is critical. Unless a new dataset dramatically changes the cache behavior, reprofiling is not necessary. In a real implementation, profiling overhead is associated with tools such as Pin and Perf, which are widely used and mature enough.

Table 4. Summary of the evaluated benchmarks.

Benchmark	CPU	GPU	HMC-atomic
Breadth-first search (`BFS`)	√	√ †	CAS if equal
Degree centrality (`DC`)	√	√	Signed add
Betweenness centrality (`BC`)	√	-	Floating-point add*
Shortest path (`SSSP`)	√	√ ‡	CAS-if equal
K-core decomposition (`KCORE`)	√	√	Signed add
Connected component (`CCOMP`)	√	-	CAS-if equal
Page rank (`PRank`)	√	√	Floating-point add*
Triangle count (`TC`)	√	-	Signed add
Graph coloring (`GC`)	√	-	CAS-if equal/-greater/less

† 5 Versions. Versions differ in their algorithmic implementation [5, 30].
‡ 4 Versions (See †).
* Single-precision floating-point addition extension (Section 2.3.4).

[6]CPU and GPU programs have different program implementations, so even if a CPU benchmark has instructions that pass the eligibility test, the corresponding GPU benchmark might not have any.

Note that after CAIRO identifies a candidate and marks its offloading region, the offloading process on both GPU and CPU is the same. In other words, when a candidate memory instruction (i.e., either a host atomic instruction or an instruction resulted from converting a group of instructions) accesses a region that is marked to be offloaded, memory controller, instead of issuing regular memory instruction to the HMC, issues the corresponding HMC-atomic instruction by modifying the type of the request. Furthermore, since HMC acknowledges the success or failure of the offloaded atomic instruction, adding dependency to such instructions are similar to regular memory requests. However, GPU workloads might fully utilize functional units in the HMC and create a performance bottleneck, but, for our evaluations, we do not observe such a bottleneck.

5.2 Evaluation of Bandwidth Sensitivity

While bandwidth-sensitive applications gain improvement in performance by increasing available bandwidth, bandwidth-insensitive applications do not, as discussed in Section 3.2. To illustrate this point (similar to Figure 2), we measure improvements in speedup when available bandwidth doubles for all workloads (from 8 to 16 GB/s), shown in Figure 7. Each marker in the figure corresponds to one workload, and the marker shape represents CPU and GPU platforms. As shown, when available bandwidth doubles, CPU workloads have significantly low bandwidth utilization, so they gain negligible speedup[7]. By contrast, when bandwidth doubles, GPU workloads have high bandwidth utilization and their performance is sensitive. Therefore, similar to Figure 2, Figure 7 shows that while CPU workloads constitute bandwidth-insensitive applications, GPU workloads constitute bandwidth-sensitive applications. For each workload, CAIRO, using micro-benchmarks and the cache profiler, identifies the type of the target application and performs analyses, illustrated in Figure 6. For the workloads in this paper, the distinction between bandwidth-sensitive and bandwidth-insensitive applications naturally divides CPU and GPU workloads. However, in exceptional cases, for an accurate estimation, CAIRO needs recognition decision from a programmer/vendor.

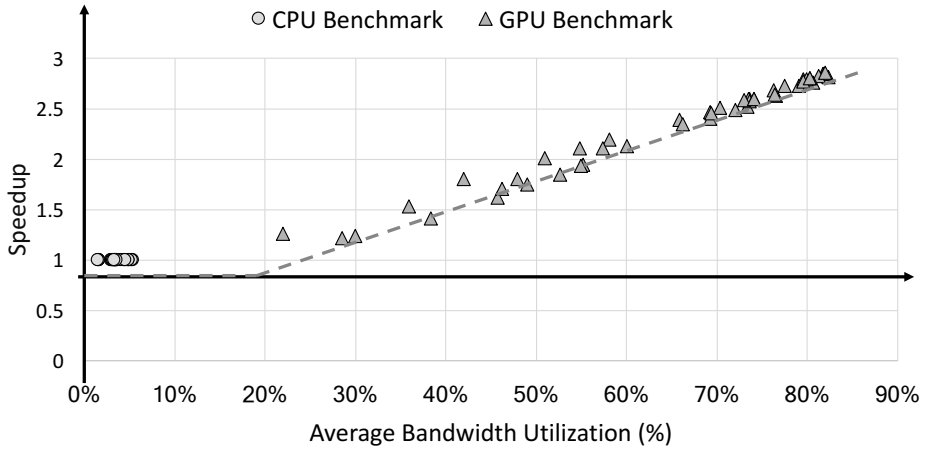

Fig. 7. Bandwidth sensitivity of CPU and GPU workloads (speedup when bandwidth is doubled from 8 to 16 GB/s; bandwidth is normalized to 16 GB/s).

5.3 Evaluation of CPU Workloads

CPU workloads, shown in Figure 7, constitute bandwidth-insensitive applications. For bandwidth-insensitive applications, Section 3 explained that offloading enables improvements in performance

[7]For more details for the breakdown of architectural behaviors of CPU workloads see Figure 2 of [26] and Figure 11 of [28].

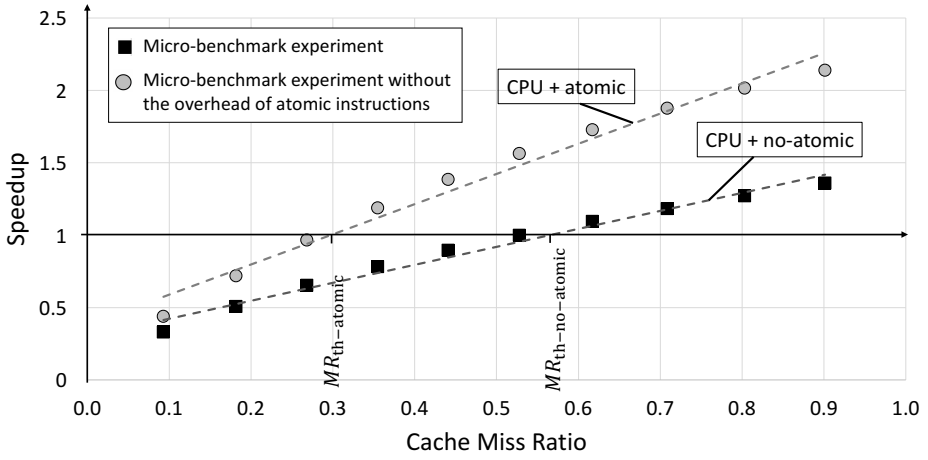

Fig. 8. Measuring machine-dependent constants with a micro-benchmark. "CPU + no-atomic" means the overhead of host atomic instructions is assumed to be zero.

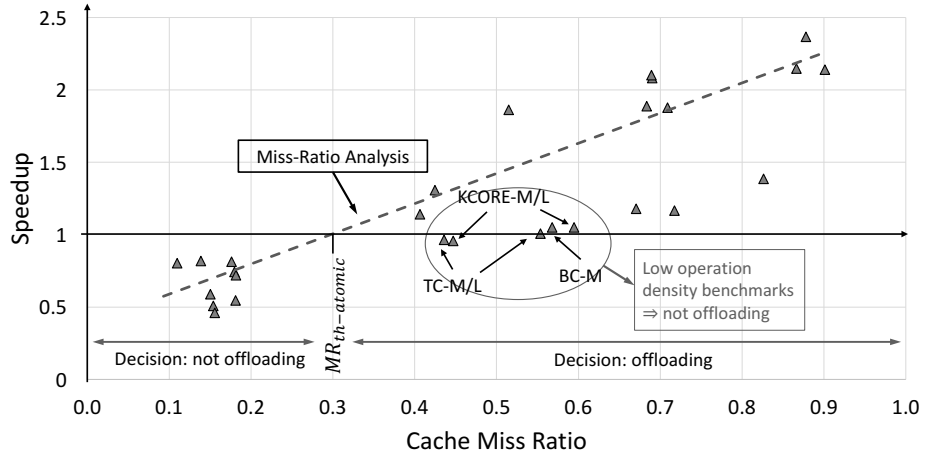

Fig. 9. CAIRO decisions for CPU workloads. The dotted line represents resulting line from linear regression from the miss-ratio analysis.

because of bypassing the cache and removing the overhead of host atomic instructions, both of which rely on the cache miss ratio of offloading candidates. To make offloading decisions, CAIRO derives machine-dependent constants in Equation 4 by performing an experiment with a micro-benchmark, which has one offloading candidate, and using a linear regression. Figure 8 shows the offloading speedup with various LLC miss ratios of the candidate in the micro-benchmark and the resulting line from the linear regression, represented by the line marked with "*CPU + atomic*", from which CAIRO derives the constants. For demonstration, we also perform the same experiment by assuming zero overhead for host atomic instructions (the line marked with "*CPU + no-atomic*"). As illustrated in the Figures 4 and 8, because offloading prevents the non-trivial overhead of host atomic instructions, the cutoff miss ratio ($MR_{\text{th-no-atomic}}$) of the no-atomic version is larger than that of the atomic version ($MR_{\text{th-atomic}}$). In our evaluation, while $\text{MR}_{\text{th-atomic}}$ remains around 30%, $\text{MR}_{\text{th-no-atomic}}$ exceeds 55%. We conclude that the overhead of host atomic instructions is one of the key factors in determining offloading decisions.

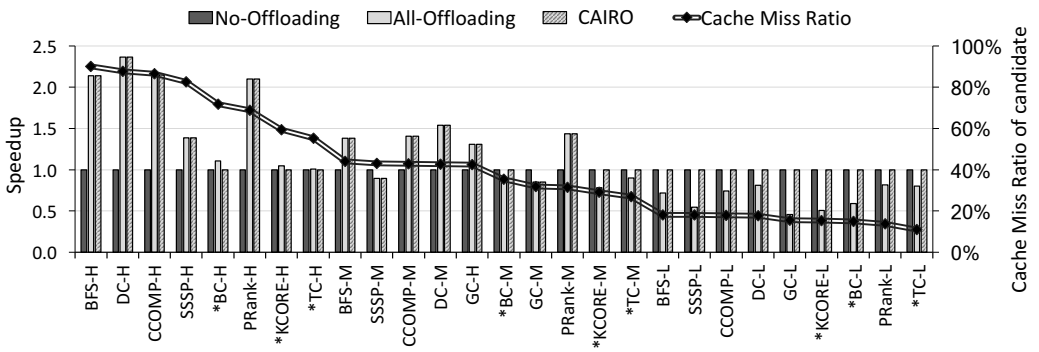

Fig. 10. Summary of CPU workloads with CAIRO ("*" marks benchmarks with low density of offloading candidates).

After deriving the machine-dependent constants for miss-ratio analysis, CAIRO performs miss-ratio analysis and makes offloading decisions for CPU, or bandwidth-insensitive workloads. To illustrate, Figure 9 shows all the workloads and the line for the miss-ratio analysis, which is the resulting line from the linear regression ("*CPU + atomic*" line in Figure 8). As shown, for workloads with a miss ratio lower than the cutoff miss ratio ($MR_{th-atomic}$), CAIRO decides to not offload their candidates. While, for workloads that their miss ratio exceed the cutoff miss ratio, CAIRO offload candidates. As pointed out in Section 5.1, to explain why CAIRO does not offload candidates with low density per memory region (i.e., those that fail CD test so they are not eligible candidates), we also evaluate workloads that contain these kind of candidates, such as `KCORE-M/L` , `TC-M/L`, and `BC-M`. In Figure 9, we observe these workloads experience almost no speedup with offloading; thus, CAIRO filters them out during the CD test.

Offloading decisions of CAIRO and resulting speedup are summarized in Figure 10. The results are sorted by the average LLC miss ratio of candidates in each workload. As mentioned in Section 5.1, for the evaluation, we generate workloads with different cache miss ratio and marked them with: "`-H`" for the original workload, "`-M`" for medium miss ratio setting, and "`-L`" for low miss ratio setting.[8] In the evaluation, we compare CAIRO with two naïve methods: (i) disabling offloading, denoted as "no-offloading" method; and (ii) offloading all eligible candidates, denoted as "all-offloading" method. We observe while "all-offloading" decision is beneficial for high miss ratio settings, it degrades performance for low miss ratio settings. By contrast, "no-offloading" decision misses the opportunity for performance speedup in the high miss ratio setting. However, CAIRO realizes the opportunity and makes an offloading decision that leads to improvement in performance. The turning point of CAIRO decisions occurs between `PRank-M` and `KCORE-M`. CAIRO offloads the candidates with higher cache miss ratios than this point, except for low density candidates, which their workloads is marked with a "*" in Figure 10. Note that since we evaluate the building blocks (i.e., kernels) of graph-computing applications, our workloads have one potential candidate in each kernel that is executed several times (e.g., see Section 6). A complex graph-computing application in a graph framework performs several traversals, computations, and updates, which is a mix of the building blocks. In Figure 10, we observe that CAIRO identifies potential candidates, and based on its characteristics makes the correct decision. In sum, for all CPU workloads, CAIRO achieves the optimized offloading decision and the close-to-optimal performance improvements.

[8]Since we are covering a broad range of candidates miss-ratio, we profile all the workloads shown in Figure 10. However, for workloads with a larger dataset than LDBC with 1M vertices (900 MB memory footprint), profiling is unnecessary. This is because the workloads behavior remain the same with a larger dataset. Although we cannot simulate the workloads with these datasets in a reasonable time, we ensured their behavior, and so CAIRO decisions, remain the same by profiling them.

5.4 Evaluation of GPU Workloads

GPU workloads constitute bandwidth-sensitive applications, as shown in Figure 7. As discussed in Section 3.2, bandwidth-sensitive applications also gain speedup from bandwidth saving. Therefore, CAIRO, after the miss-ratio analysis, to make the offloading decision, performs the BW-saving analysis.

Section 4.2 explained that after miss-ratio analysis, CAIRO divides miss ratio range to three decision regions. Figure 11 shows the miss-ratio analysis for GPU workloads. To conservatively mark the decision regions, we perform miss-ratio analysis on two workloads at the two ends of the spectrum (i.e., the spectrum in which the benefits of offloading for `BFS-ttc` starts at the smallest cache miss ratio, while same benefits for `PRank` starts at the highest cache miss ratio). Based on our findings, for $MissRatio_H$ and $MissRatio_L$ we use conservative values of 30% and 80%, respectively. The low miss ratio region always hurts performance, and the high miss ratio region always improves

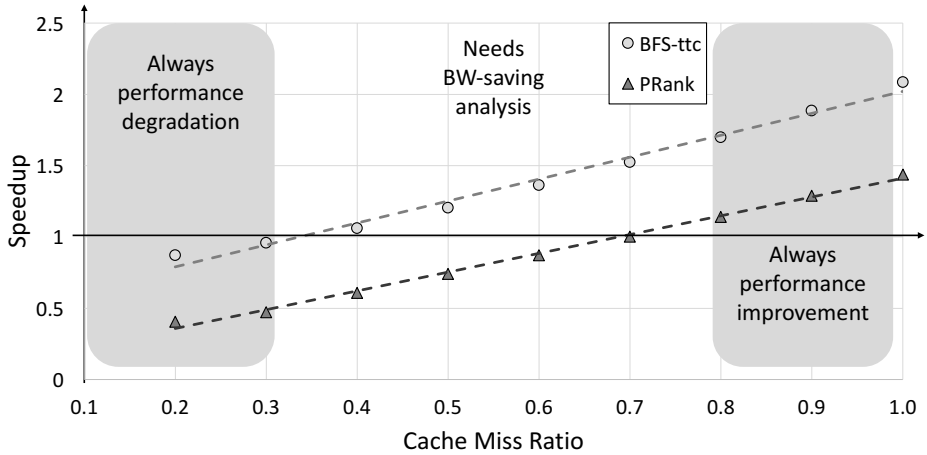

Fig. 11. Miss-ratio analysis for `BFS-ttc` and `PRank`.

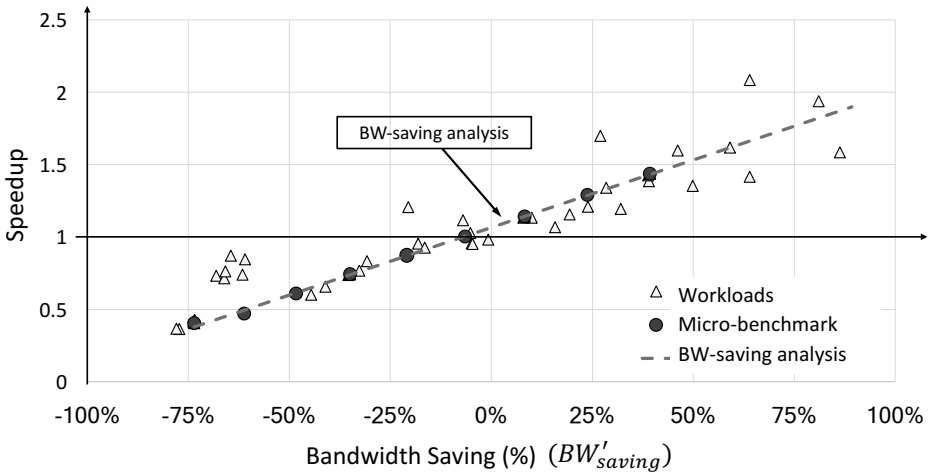

Fig. 12. CAIRO decisions for GPU workloads. The dotted line represents resulting line from linear regression of the BW-saving analysis.

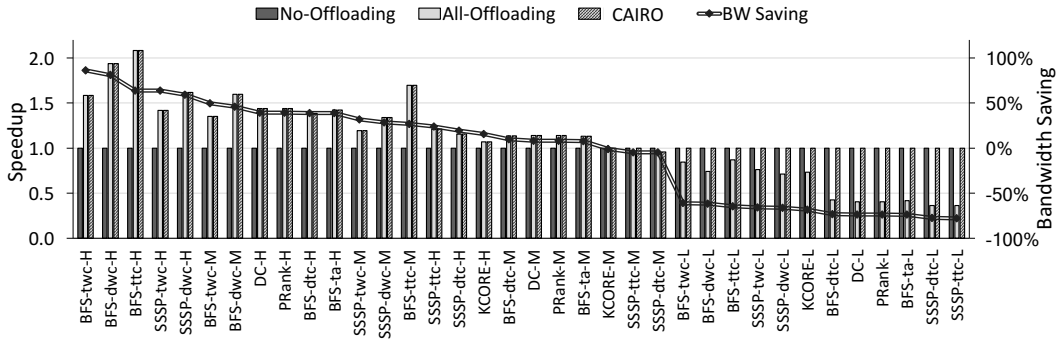

Fig. 13. Summary of GPU workloads with CAIRO (extra BW-saving analysis step is done for benchmarks with "-M").

performance. For the middle region, because of the uncertainty in the speedup result from the miss-ratio analysis, BW-saving analysis is necessary. CAIRO estimates the speedup of bandwidth-sensitive applications by driving machine-dependent constants in Equation 6 by performing an experiment with a micro-benchmark and using a linear regression. Figure 12 illustrates the micro-benchmark experiment, resulting line from the linear regression, and all the workloads in a single graph. We derive machine-dependent constants of M_1 and M_2 in Equation 6 from dotted line in the figure. The figure shows that bandwidth-sensitive applications achieve higher improvement in performance with more bandwidth savings. By utilizing bandwidth savings as a key metric, CAIRO differentiates workloads and makes the appropriate offloading decision.

Figure 13 summarizes applying CAIRO to GPU workloads, which is sorted according to bandwidth savings of each workload when we offload their candidates. Note that each kernel in GPU version have multiple algorithmic implementation, in which program behavior is not the same [5, 30] (e.g., for `BFS`:`BFS-twc`, `BFS-dwc`, or `BFS-ttc` [5, 28]). Similar to the CPU evaluation, we compare our technique with two methods: disabling offloading (no-offloading method), and offloading all eligible candidates (all-offloading method). As results show, although different benchmarks have different bandwidth sensitivity, higher bandwidth savings enables more improvements in performance. Therefore, on the low bandwidth-savings side of the figure, "all-offloading" method causes degradation in performance from unnecessary offloading, while at the high bandwidth-savings side, "no-offloading" method misses the opportunity for performance improvements. However, CAIRO achieves the optimized offloading decision and performance for the workloads. Note that, for CPU workloads, although offloading enables bandwidth savings, CPU workloads do not gain speedup from it, discussed in Figure 7. CPU workloads gain the most of their speedup from bypassing the cache and avoiding the overhead of host atomic instructions, whereas, GPU workloads gain benefit from bandwidth savings, besides mentioned sources.

6 A CASE STUDY

Figures 14 and 15 show code snippets of the *breadth-first search* (`BFS`) algorithm [28] for GPUs and CPUs respectively [27]. The algorithm traverses a given graph and visits all of its vertices once while checking and updating their properties (*graph property* [27]). The graph traversal code goes through a loop that iterates over its steps in a synchronized way. For the GPU version, the application creates an initial frontier list of vertices and launches a CUDA kernel for each BFS step. Then, it starts a thread for each vertex and if the vertex is in the frontier, its neighbors will be checked and updated. Meanwhile, any unseen neighbor is added to the next frontier. In this algorithm, since all

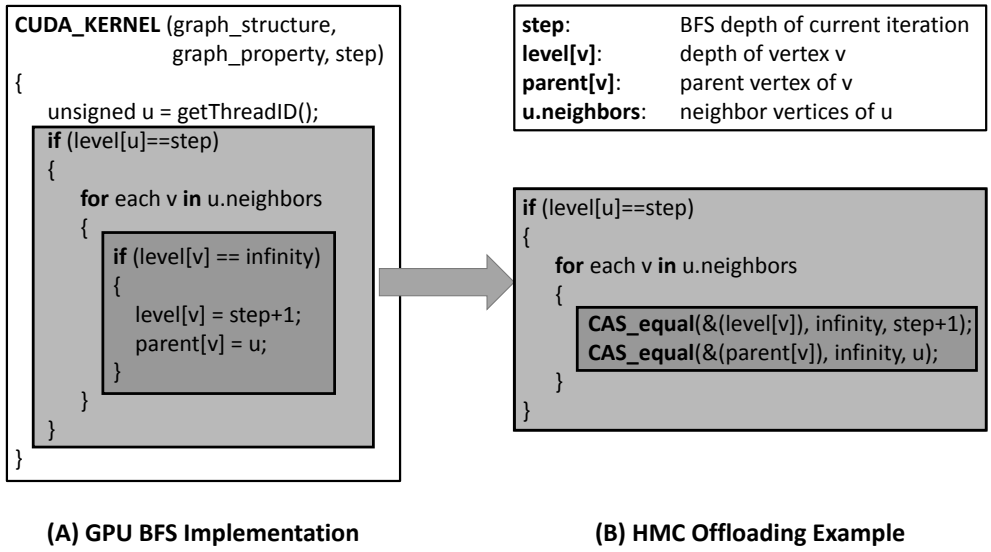

Fig. 14. The GPU version of instruction offloading for BFS.

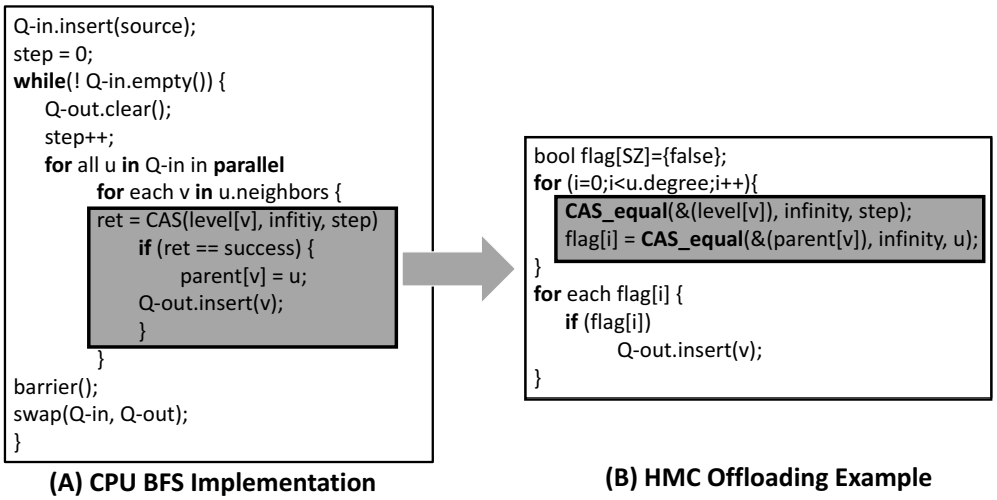

Fig. 15. The CPU version of instruction offloading for BFS.

threads access vertices concurrently, the bandwidth requirement is significantly high. The traversing of the neighbor list, which accesses graph property, has a low cache hit ratio besides high memory bandwidth requirements. Hence, the code for checking the graph property is a good candidate for offloading. In fact, each thread generates two HMC-atomic instructions. For each thread, the generated HMC-atomic instructions are data independent; therefore, their execution is overlapped. Similar to the GPU case, for the CPU case the statements modifying the graph property are offloaded to the memory. However, in the CPU case, HMC-atomic instruction is inserted by converting a generic host atomic instruction. Moreover, statements following the host atomic instruction are also

offloaded because of low cache miss ratio. In the CPU case, the second HMC-atomic instruction is dependent on the first one. Therefore, the second HMC-atomic instruction is executed only after the completion of the first HMC-atomic instruction. Although this dependency between HMC-atomic instructions limits their concurrent execution, it prevents the overhead of host atomic instructions.

7 RELATED WORK

Processing-in-memory was initiated decades ago with several research proposals and fabricated chips [13, 15, 20, 31, 34]. However, because of the fabrication difficulties and less immediate needs, PIM was not adopted widely by the industry. Recent advances in 3D-stacking technology have reignited interest in PIM. For instance, multiple industry designs such as Hybrid Memory Cube (HMC) [6, 19, 35], High Bandwidth Memory (HBM) [23, 25], and Active Memory Cube (AMC) [29] have been developed. Moreover, academia have started to investigate PIM [2–4, 7, 12, 17, 21, 22, 26, 44]. For instance, in PEI [3], Ahn et al. proposed a fixed-function PIM for CPU. To support a set of PIM operations, they add custom-hardware deigns to both host processors and memory hierarchies. In PEI, according to a runtime locality monitoring in the custom-hardware, PIM operations will either be processed in the host processor or be offloaded to the memory. CAIRO exceeds performance gain of PEI with less complexity for CPU benchmarks. Nai et al. discussed this gain in more details in their GraphPIM paper [26]. In TOP-PIM [44], Zhang et al. utilized APUs as both host processors and PIM processors. TOP-PIM builds a performance prediction methodology for GPGPU kernels when they are offloaded to PIM using machine-learning models. While TOP-PIM studies kernel-level offloading to PIM units that have the same computation power and features, CAIRO studies instruction-level offloading to PIM units that have simple computational units, based on the HMC 2.0 specification. Nai et al. in GraphPIM [26] followed the HMC specification and selected instruction-level offloading candidates manually for a set of graph-computing benchmarks on CPU. However, they did not propose any mechanisms that determine offloading candidates and their selection for GPU applications.

Despite numerous proposals about PIM architectures, little effort has been invested in compiler support for instruction-level offloading of PIM. Most recent research proposals rely on skills of programmers to properly utilize the PIM functionality. However, as both software and PIM architecture become more and more complex, such a task will become more sophisticated and error-prone. To the best of our knowledge, the most relevant work about identifying characteristics for offloading targets is transparent offloading and mapping (TOM) [17]. Hsieh et al. studied how to efficiently offload code sections (i.e., kernel-level offloading) in a system with multiple 3D-stacked memories, whereas CAIRO studies instruction-level offloading. In fact, Hsieh et al. provided a straightforward compiler-based analysis based on cost-benefit calculations that measures potential bandwidth savings in offloading a block of data. TOM focuses more on the benefits enabled by bandwidth savings of full-programmable PIM cores. Thus, it is less applicable for instruction-level PIM offloading and it inherently different than CAIRO. CAIRO, beside bandwidth-savings analysis, introduces the overhead of host atomic instruction and density of the candidates in the offloading decision, and provides a systematic way for identifying offloading candidates.

8 CONCLUSION

Numerous hardware studies have been devoted to PIM research because of the recent advances in 3D-stacking technology, which has motivated various hardware proposals and designs. However, few studies have focused on compiler support for PIM or recognized the benefits of instruction-level offloading. To address this research gap, we proposed CAIRO, a technique that selects instruction-level offloading candidates in the context of soon-to-be commercially available HMC 2.0. Although

we showcase HMC for evaluations, CAIRO targets enabling generic instruction-level offloading for PIM. This is because, in CAIRO, only the eligibility test depends on a PIM architecture specifications. CAIRO, which investigates and models benefits of instruction-level offloading, is composed of a cache profiler, a compile-time analysis phase, and benefit analysis models. The key factors in offloading decisions are the cache miss ratio of candidates, the bandwidth savings of offloading, and the density of offloading candidates per memory region. By evaluating 63 workloads from graph-computing applications (27 for CPU and 36 for GPU) and measuring improvements in the performance of CAIRO over that of two naïve methods of always-offloading and no-offloading cases, we successfully demonstrated the effectiveness of this technique.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable comments. We also thank Sandia National Labs for providing the SST/VaultSim framework. We gratefully acknowledge the support of National Science Foundation CCF-1337177 and CCF-1533767.

REFERENCES

[1] 2017. MacSim. http://github.com/gthparch/macsim. (2017).

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 105–117.

[3] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 336–348.

[4] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. 2016. Design and Evaluation of a Processing-in-Memory Architecture for the Smart Memory Cube. In *ARCS 2016: 29th International Conference on Architecture of Computing Systems*. Springer International Publishing, 19–31.

[5] Nagesh B Lakshminarayana. 2014. *Efficient Graph Algorithm Execution on Data-Parallel Architectures*. Ph.D. Dissertation. Georgia Institute of Technology.

[6] Bryan Black. 2013. Die Stacking is Happening. In *Microarchitecture (MICRO), 2013 46rd Annual IEEE/ACM International Symposium on*. Keynote I Presentation.

[7] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Kevin Hsieh, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu. 2017. LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory. *IEEE Computer Architecture Letters (CAL)* PP, 99 (May 2017).

[8] HMC Consortium. 2014. Hybrid Memory Cube Specification 2.0. *Retrieved from hybridmemorycube.org* (2014). [Online; accessed 30-March-2017].

[9] Xiangyu Dong and Yuan Xie. 2009. System-level Cost Analysis and Design Exploration for Three-dimensional Integrated Circuits (3D ICs). In *Proceedings of the 2009 Asia and South Pacific Design Automation Conference (ASP-DAC '09)*. IEEE Press, Piscataway, NJ, USA, 234–241.

[10] Marwa Elteir, Heshan Lin, and Wu-chun Feng. 2011. Performance Characterization and Optimization of Atomic Operations on AMD GPUs. In *2011 IEEE International Conference on Cluster Computing*. IEEE Press, Piscataway, NJ, USA, 234–243.

[11] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 619–630.

[12] David Fick, Ronald G Dreslinski, Bharan Giridhar, Gyouho Kim, Sangwon Seo, Matthew Fojtik, Sudhir Satpathy, Yoonmyung Lee, Daeyeon Kim, Nurrachman Liu, and others. 2012. Centip3De: A 3930DMIPS/W configurable near-threshold 3D stacked system with 64 ARM Cortex-M3 cores. In *2012 IEEE International Solid-State Circuits Conference*. IEEE Press, Piscataway, NJ, USA, 190–192.

[13] Maya Gokhale, Bill Holmes, and Ken Iobst. 1995. Processing in Memory: the Terasys Massively Parallel PIM Array. *Computer* 28, 4 (Apr 1995), 23–31.

[14] Gabriel H., Loh Nuwan Jayasena, Mark H. Oskin, Mark Nutter, David Roberts, Mitesh Meswani, Dong Ping Zhang, and Mike Ignatowski. 2013. A Processing-in-Memory Taxonomy and a Case for Studying Fixed-function PIM. In *WoNDP: 1st Workshop on Near-Data Processing*. IEEE Press, Piscataway, NJ, USA.

[15] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Jay Brockman, Apoorv Srivastava, William Athas, Vincent Freeh, Jaewook Shin, and Joonseok Park. 1999. Mapping Irregular Applications to DIVA, a PIM-based Data-intensive Architecture. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (SC '99)*. ACM, New York, NY, USA.

[16] HMC Consortium. 2013. Hybrid Memory Cube Specification 1.1. *Retrieved from hybridmemorycube.org* (2013). [Online; accessed 30-March-2017].

[17] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W. Keckler. 2016. Transparent Offloading and Mapping (TOM): Enabling Programmer-transparent Near-data Processing in GPU Systems. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 204–216.

[18] Intel. 2015. *Intel 64 and IA-32 Architectures Software Developer's Manual*. San Francisco, CA, USA.

[19] Joe Jeddeloh and Brent Keeth. 2012. Hybrid Memory Cube New DRAM Architecture Increases Density and Performance. In *VLSI Technology (VLSIT), 2012 Symposium on*. IEEE Press, Piscataway, NJ, USA, 87–88.

[20] Yi Kang, Wei Huang, Seung-Moon Yoo, Diana Keen, Zhenzhou Ge, Vinh Lam, Pratap Pattnaik, and Josep Torrellas. 2012. FlexRAM: Toward an Advanced Intelligent Memory System. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*. IEEE Press, Piscataway, NJ, USA, 5–14.

[21] Dae Hyun Kim, Krit Athikulwongse, Michael B. Healy, Mohammad M. Hossain, Moongon Jung, Ilya Khorosh, Gokul Kumar, Young-Joon Lee, Dean L. Lewis, Tzu-Wei Lin, Michael Wieckowski, Gregory Chen, Trevor Mudge, Dennis Sylvester, and david Blaauw. 2015. Design and Analysis of 3D-MAPS (3D Massively Parallel Processor with Stacked Memory). *IEEE Trans. Comput.* 64, 1 (Jan 2015), 112–125.

[22] Gwangsun Kim, John Kim, Jung Ho Ahn, and Jaeha Kim. 2013. Memory-centric System Interconnect Design with Hybrid Memory Cubes. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13)*. IEEE Press, Piscataway, NJ, USA, 145–156.

[23] Joonyoung Kim and Younsu Kim. 2014. HBM: Memory solution for bandwidth-hungry processors. In *2014 IEEE Hot Chips 26 Symposium (HCS)*. IEEE Press, Piscataway, NJ, USA, 1–24.

[24] Pranith Kumar, Lifeng Nai, and Hyesoon Kim. 2016. Analyzing Consistency Issues in HMC Atomics. In *Proceedings of the Second International Symposium on Memory Systems (MEMSYS '16)*. ACM, New York, NY, USA, 151–152.

[25] Dong Uk Lee, Kyung Whan Kim, Kwan Weon Kim, Hongjung Kim, Ju Young Kim, Young Jun Park, Jae Hwan Kim, Dae Suk Kim, Heat Bit Park, Jin Wook Shin, and others. 2014. 25.2 A 1.2V 8Gb 8-channel 128GB/s High-Bandwidth Memory (HBM) Stacked DRAM with Effective Microbump I/O Test Methods Using 29nm Process and TSV. In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC '14)*. IEEE Press, Piscataway, NJ, USA, 432–433.

[26] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *2017 IEEE 23st International Symposium on High Performance Computer Architecture (HPCA '17')*.

[27] Lifeng Nai and Hyesoon Kim. 2015. Instruction Offloading with HMC 2.0 Standard: A Case Study for Graph Traversals. In *Proceedings of the 2015 International Symposium on Memory Systems (MEMSYS '15)*. ACM, New York, NY, USA, 258–261.

[28] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, 69:1–69:12. https://github.com/graphbig/GraphBIG-Doc/blob/master/HOWTO/HOWTO-run.md

[29] Ravi Nair, Samuel F. Antao, Carlo Bertolli, Pradip Bose, Jose R. Brunheroto, Tong Chen, Chen-Yong Cher, Carlos H. A. Costa, Constantinos Evangelinos, Bruce M. Fleischer, Thomas W. Fox, Diego S. Gallo, Leopold Grinberg, John A. Gunnels, Arpith C. Jacob, Philip Jacob, Hans M. Jacobson, Tejas Karkhanis, C. Kim, Jaime H. Moreno, J. Kevin O'Brien, Martin Ohmacht, Yoonho Park, Daniel A. Prener, Bryan S. Rosenburg, Kyung Dong Ryu, Olivier Sallenave, Mauricio J. Serrano, Patrick D. M. Siegl, Krishnan Sugavanam, and Zehra Sura. 2015. Active Memory Cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development* 59, 2/3 (March 2015), 17:1–17:14.

[30] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Data-Driven Versus Topology-driven Irregular Computations on GPUs. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE Press, Piscataway, NJ, USA, 463–474.

[31] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. 1998. Active Pages: A Computation Model for Intelligent Memory. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*. IEEE Press, Piscataway, NJ, USA, 192–203.

[32] David Patterson. 2004. Why Latency Lags Bandwidth, and What it Means to Computing. In *Keynote address ,Workshop on High Performance Embedded Computing (ISCA '16)*.

[33] David Patterson. 2009. The Top 10 Innovations in the New NVIDIA Fermi Architecture, and The Top 3 Next Challenges. *NVIDIA Whitepaper* 47 (2009).

[34] David Patterson, Krste Asanovic, Aaron Brown, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, Christoforos Kozyrakis, David Martin, Stylianos Perissakis, Randi Thomas, Noah Treuhaft, and Katherine Yelick. 1997. Intelligent RAM (IRAM): the Industrial Setting, Applications, and Architectures. In *Proceedings International Conference on Computer Design VLSI in Computers and Processors*. IEEE Press, Piscataway, NJ, USA, 2–7.

[35] Thomas Pawlowski. 2011. Hybrid Memory Cube (HMC). In *Hot Chips 23 Symposium (HCS), 2011 IEEE*. IEEE Press, Piscataway, NJ, USA, 1–24.

[36] Peter Ramm, Armin Klumpp, Josef Weber, Nicolas Lietaer, Maaike Taklo, Walter De Raedt, Thomas Fritzsch, and Pascal Couderc. 2010. 3D Integration technology: Status and Application Development. In *2010 Proceedings of ESSCIRC*. IEEE Press, Piscataway, NJ, USA, 9–16.

[37] Vijay Janapa Reddi, Alex Settle, Daniel A. Connors, and Robert S. Cohn. 2004. PIN: A Binary Instrumentation Tool for Computer Architecture Research and Education. In *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture (WCAE '04)*. ACM, New York, NY, USA.

[38] Arun Rodrigues, Scott Hemmert, Brian W. Barrett, Chad Kersey, Ron Oldfield, Marlo Weston, Rolf Risen, Jeanine Cook, Paul Rosenfeld, Elliot Cooper-Balis, and Bruce Jacob. 2011. The Structural Simulation Toolkit. *SIGMETRICS Perform. Eval. Rev.* 38, 4 (March 2011), 37–42.

[39] Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. 2009. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. *SIGARCH Comput. Archit. News* 37, 3 (June 2009), 371–382.

[40] Paul Rosenfeld. 2014. *Performance Exploration of the Hybrid Memory Cube*. Ph.D. dissertation. University of Maryland, College Park.

[41] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* 10, 1 (Jan 2011), 16–19.

[42] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. 2015. Evaluating the Cost of Atomic Operations on Modern Architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE Press, Piscataway, NJ, USA, 445–456.

[43] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News* 23, 1 (March 1995), 20–24.

[44] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: Throughput-oriented Programmable Processing in Memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC '14)*. ACM, New York, NY, USA, 85–98.